Refining Abstract Interpretation Based Value Analysis with Constraint Programming Techniques

Olivier Ponsini, Claude Michel, Michel Rueher

Accepted for publication at CP'12 (18th International Conference on Principles and Practice of Constraint Programming)

Refining abstract interpretation based value analysis with constraint programming techniques*

Olivier Ponsini, Claude Michel, and Michel Rueher

University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
firstname.lastname@unice.fr

Abstract. Abstract interpretation based value analysis is a classical approach for verifying programs with floating-point computations. However, state-of-the-art tools compute an over-approximation of the variable values that can be very coarse. In this paper, we show that constraint solvers can significantly refine the approximations computed with abstract interpretation tools. We introduce a hybrid approach that combines abstract interpretation and constraint programming techniques in a single static and automatic analysis. rAiCp, the system we developed is substantially more precise than Fluctuat, a state-of-the-art static analyser. Moreover, it could eliminate 13 false alarms generated by Fluctuat on a standard set of benchmarks.

Key words: Program verification, Floating-point computation, Constraint solvers over floating-point numbers, Constraint solvers over real number intervals, Abstract interpretation-based approximation

1 Introduction

Programs with floating-point computations control complex and critical physical systems in various domains such as transportation, nuclear energy, or medicine. Floating-point computations are an additional source of errors and famous computer bugs are due to errors in floating-point computations, e.g., the Patriot missile failure. Floating-point computations are usually derived from mathematical models on real numbers [14]. However, real and floating-point computation models are different: for the same sequence of operations, floating-point numbers do not behave identically to real numbers. For instance, with binary floating-point numbers, some decimal real numbers are not representable (e.g., 0.1 has no exact representation), arithmetic operators are not associative and may be subject to phenomena such as absorption (e.g., $a + b$ is rounded to a when a is far greater than b) or cancellation (subtraction of nearly equal operands after rounding that only keeps the rounding error).

* This work was partially supported by ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), and OSEO ISI PAJERO projects.

Value analysis is often used to check the absence of run-time errors, such as invalid integer or floating-point operations, as well as simple user assertions [8]. Value analysis can also help with estimating the accuracy of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers. Existing automatic tools are mainly based on abstract interpretation techniques. For instance, Fluctuat [9], a state-of-the-art static analyzer, computes an over-approximation of the domains of the variables for a C program considered with a semantics on real numbers. It also computes an over-approximation of the error due to floating-point operations at each program point. However, these over-approximations may be very coarse even for usual programming constructs and expressions. As a consequence, numerous false alarms[1]—also called false positives—may be generated.

In this paper, we introduce a hybrid approach for the value analysis of floating-point programs that combines abstract interpretation (AI) and constraint programming techniques (CP). We show that constraint solvers over floating-point and real numbers can significantly refine the over-approximations computed by abstract interpretation. rAiCp, the system we developed, uses both Fluctuat and the following constraint solvers:

- RealPaver [17], a safe and correct solver for constraints over real numbers,
- FPCS [21, 20], a safe and correct solver for constraints over floating-point numbers.

Experiments show that rAiCp is substantially more precise than Fluctuat, especially on C programs that are difficult to handle with abstract interpretation techniques. This is mainly due to the refutation capabilities of filtering algorithms over the real numbers and the floating-point numbers used in rAiCp. rAiCp could also eliminate 13 false alarms generated by Fluctuat on a set of 57 standard benchmarks proposed by D'Silva et al [12] to evaluate CDFL, a program analysis tool that embeds an abstract domain in the conflict driven clause learning algorithm of a SAT solver. Moreover, rAiCp is on average at least 5 times faster than CDFL on this set of benchmarks.

Section 2 illustrates our approach on a small example. Basics on the techniques and tools we use are introduced in Section 3. Next section is devoted to related work. Section 5 details our approach whereas experiments are analysed in Section 6.

2 Motivation

In this section, we illustrate our approach on a small example. The program in Fig. 1 is mentioned in [13] as a difficult program for abstract interpretation based

[1] A false alarm corresponds to the case when the abstract semantics intersects the forbidden zone, i.e., erroneous program states, while the concrete semantics does not intersect this forbidden zone. So, a potential error is signaled which can never occur in reality (see `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`).

Fig. 1. Example 1.

```
1 /* Pre−condition :  x ∈ [0, 10] */
2 double conditional(double x) {
3   double y = x*x - x;
4   if (y >= 0)
5     y = x/10;
6   else
7     y = x*x + 2;
8   return y; }
```

analyses. On floating-point numbers, as well as on real numbers, this function returns a value in the interval $[0, 3]$. Indeed, from the conditional statement of line 4, we can derive the following information:

- `if` branch: $x = 0$ or $x \geq 1$, and thus $y \in [0, 1]$ at the end of this branch;
- `else` branch: $x \in]0, 1[$, and thus $y \in]2, 3[$ at the end of this branch.

However, classical abstract domains (e.g., intervals, polyhedra), as well as the abstract domain of *zonotopes* used in FLUCTUAT, fail to obtain a good approximation of this value. The best interval obtained with these abstractions is $[0, 102]$, both over the real numbers and the floating-point numbers. The difficulty for these analyses is to intersect the abstract domains computed for y at lines 3 and 4. Actually, they are unable to derive from these statements any constraint on x. As a consequence, in the `else` branch, they still estimate that x ranges over $[0, 10]$.

We propose here to compute an approximation of the domains in both execution paths. On this example, CSP filtering techniques are strong enough to reduce the domains of the variables. Consider for instance the constraint system over the real numbers $\{y_0 = x_0 * x_0 - x_0, y_0 < 0, y_1 = x_0 * x_0 + 2, x_0 \in [0, 10]\}$ which corresponds to the execution path[2] through the `else` branch of the function `conditional`. From the constraints $y_0 = x_0 * x_0 - x_0$ and $y_0 < 0$, the interval solver over the real numbers we use can reduce the initial domain of x_0 to $[0, 1]$. This reduced domain is then used to compute the one of y_1 *via* the constraint $y_1 = x_0 * x_0 + 2$, which yields $y_1 \in [2, 3.001]$. Likewise, our constraint solver over the floating-point numbers will reduce x_0 to $[4.94 \times 10^{-324}, 1.026]$ and y_1 to $[2, 3.027]$.

To sum up, we explore the control flow graph (CFG) of a program and stop each time two branches join. There, we build one constraint system per branch that reaches the join point. Then, we use filtering techniques on these systems to reduce the domains of the variables computed by FLUCTUAT at this join point. Exploration goes on with the reduced domains. CFG exploration is performed on-the-fly. Branches are cut as soon as an inconsistency of the constraint system

[2] Statements are converted into DSA (Dynamic Single Assignment) form where each variable is assigned exactly once on each program path [2].

Table 1. Return domain of the `conditional` function.

	Domain	Time
Exact real and floating-point domains	$[0, 3]$	–
FLUCTUAT (real and floating-point domains)	$[0, 102]$	0.1 s
FPCS (floating-point domain)	$[0, 3.027]$	0.2 s
REALPAVER (real domain)	$[0, 3.001]$	0.3 s

is detected by a local filtering algorithm. Table 1 collects the results obtained by the different techniques on the example of the function `conditional`. On this example, contrary to FLUCTUAT, our approach computes very good approximations. Analysis times are very similar. In [13], the authors proposed an extension to the zonotopes—named *constrained zonotopes*—which attempts to overcome the issue of program conditional statements. This extension is defined for the real numbers and is not yet implemented in FLUCTUAT. The approximation computed with *constrained zonotopes* is better than the one of FLUCTUAT (the upper bound is reduced to 9.72) but remains less precise than the one computed with REALPAVER.

3 Background

Before going into the details, we recall basics on abstract interpretation and FLUCTUAT, as well as on the constraint solvers REALPAVER and FPCS used in our implementation.

Abstract interpretation[3] consists in considering an abstract semantics, that is a super-set of the concrete program semantics. The abstract semantics covers all possible cases, thus, if the abstract semantics is safe (i.e. does not intersect the forbidden zone) then so is the concrete semantics.

FLUCTUAT is a static analyzer for C programs specialized in estimating the precision of floating-point computations[4] [9]. FLUCTUAT compares the behavior of the analyzed program over real numbers and over floating-point numbers. In other words, it allows to specify ranges of values for the program input variables and computes for each program variable v:

- bounds for the domain of variable v considered as a real number;
- bounds for the domain of variable v considered as a floating-point number;
- bounds for the maximum error between real and floating-point values;
- the contribution of each statement to the error associated with variable v ;

[3] See `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html` for a nice informal introduction.

[4] FLUCTUAT is developed by CEA-LIST (`http://www-list.cea.fr/validation_en.html`) and was successfully used for industrial applications of several tens of thousands of lines of code in transportation, nuclear energy, or avionics areas.

– the contribution of the input variables to the error associated with variable v.

FLUCTUAT proceeds by abstract interpretation. It uses the weakly relational abstract domain of zonotopes [15]. Zonotopes are sets of affine forms that preserve linear correlations between variables. They offer a good trade-off between performance and precision for floating-point and real number computations. Indeed, the analysis is fast and scales well, processes accurately linear expressions, and keeps track of the statements involved in the loss of accuracy of floating-point computations. To increase the analysis precision, FLUCTUAT allows to use arbitrary precision numbers or to subdivide up to two input variable intervals. However, over-approximations computed by FLUCTUAT may be very large because the abstract domains do not handle well conditional statements and non-linear expressions.

REALPAVER is an interval solver for numerical constraint systems over the real numbers[5] [17]. Constraints can be non-linear and can contain the usual arithmetic operations and transcendental elementary functions.

REALPAVER computes reliable approximations of continuous solution sets using correctly rounded interval methods and constraint satisfaction techniques. More precisely, the computed domains are closed intervals bounded by floating-point numbers. REALPAVER implements several partial consistencies: box, hull, and $3B$ consistencies. An approximation of a solution is described by a box, i.e., the Cartesian product of the domains of the variables. REALPAVER either proves the unsatisfiability of the constraint system or computes small boxes that contains all the solutions of the system.

The REALPAVER modeling language does not provide strict inequality and not-equal operators, which can be found in conditional expressions in programs. As a consequence, in the constraint systems generated for REALPAVER, strict inequalities are replaced by non strict ones and constraints with a not-equal operator are ignored. This may lead to over-approximations, but this is safe since no solution is lost.

FPCS is a constraint solver designed to solve a set of constraints over floating-point numbers without losing any solution [21, 20]. It uses $2B$-consistency [19] along with projection functions adapted to floating-point arithmetic [22, 4].

The main difficulty lies in computing inverse projection functions that keep all the solutions. Indeed, direct projections only requires a slight adaptation of classical results on interval arithmetic, but inverse projections do not follow the same rules because of the properties of floating-point arithmetic. More precisely, each constraint is decomposed into an equivalent binary or ternary constraint by introducing new variables if necessary. A ternary constraint $x = y \odot_f z$, where $\odot_f$ is an arithmetic operator over the floating-point numbers, is decomposed into three projection functions:

[5] REALPAVER web site: `http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/`

– the direct projection, $\Pi_x(x = y \odot_f z)$;
– the first inverse projection, $\Pi_y(x = y \odot_f z)$;
– the second inverse projection, $\Pi_z(x = y \odot_f z)$.

A binary constraint of the form $x \odot_f y$, where $\odot_f$ is a relational operator (among `==`, `!=`, `<`, `<=`, `>`, and `>=`), is decomposed into two projection functions: $\Pi_x(x \odot_f y)$ and $\Pi_y(x \odot_f y)$. The computation of the approximation of these projection functions is mainly inspired from interval arithmetic and benefits from floating-point numbers being a totally ordered finite set.

FPCS also implements stronger consistencies—e.g., kB-consistencies [19]—to deal with the classical issues of multiple occurrences and to reduce more substantially the bounds of the domains of the variables.

The floating-point domains handled by FPCS also include infinities. Moreover, FPCS handles all the basic arithmetic operations, as well as most of the usual mathematical functions. Type conversions are also correctly processed.

4 Related work

Different methods address static validation of programs with floating-point computations: abstract interpretation based analyses, proofs of programs with proof assistants or with decision procedures in automatic solvers.

Analyses based on abstract interpretation capture rounding errors due to floating-point computation in their abstract domains. They are usually fast, automatic, and scalable. However, they may lack of precision and they are not tailored for automatically generating a counter-example, that is to say, input variable values that violate some assertion in a program. Astrée [8] is probably one of the most famous tool in this family of methods. The tool estimates the value of the program variables at every program point and can show the absence of run-time errors, that is the absence of behavior not defined by the programming language, e.g., division by zero, arithmetic overflow. As said before, Fluctuat estimates in addition the accuracy of the floating-point computations, that is, a bound on the difference between the values taken by variables when the program is given a real semantics and when it is given a floating-point semantics [9].

Proof assistants like Coq [3] or HOL [18] allow their users to formalize floating-point arithmetic. Proofs of program properties are done manually in the proof assistants which guarantee proof correctness. Even though some parts of the proofs may be automatized, these tools usually require a lot of user interaction. Moreover, when a proof strategy fails to prove a property, the user often does not know whether the property is false or another strategy could prove it. Like abstract interpretation, proof assistants usually do not provide automatic generation of counter-examples for false properties. The Gappa tool [11] combines interval arithmetic and term rewriting from a base of theorems. The theorems rewrite arithmetic expressions so as to compensate for the shortcomings of interval arithmetic, e.g., loss of dependency between variables. Whenever

the computed intervals are not precise enough, theorems can be manually introduced or the input domains can be subdivided. The cost of this semi-automatic method is then considerable. In [1], the authors propose axiomatizing floating-point arithmetic within first-order logic to automate the proofs conducted in proof assistants such as Coq by calling external SMT (Satisfiability Modulo Theories) solvers and Gappa. Their experiments show that human interaction with the proof assistant is still required.

The classical bit-vector approach of SAT solvers is ineffective on programs with floating-point computations because of the size of the domains of floating-point variables and the cost of bit-vector operations. An abstraction technique was devised for CBMC in [5]. It is based on under and over-approximation of floating-point numbers with respect to a given precision expressed as a number of bits of the mantissa. However, this technique remains slow. D'Silva et al [12] developed recently CDFL, a program analysis tool that embeds an abstract domain in a conflict driven clause learning algorithm of a SAT solver. CDFL is based on a sound and complete analysis for determining the range of floating-point variables in control software. In [12] the authors state that CDFL is more than 200 times faster than CBMC. In Section 6 we compare the performances of CDFL and rAiCp on a set of benchmarks proposed by D'Silva et al.

Links between abstract interpretation and constraint logic programming have been studied at a theoretical level (e.g., [6]) and recent work investigate the use of abstract interpretation and abstract domains in the context of constraint programming. In [10], the authors introduce a new global constraint to model iterative arithmetic relations between integer variables. The associated filtering algorithm is based on abstract interpretation over polyhedra. In [23], the authors propose to use the octagonal abstract domain, which proved efficient in abstract interpretation, to represent the variable domains in a continuous constraint satisfaction problem. Then, they generalize local consistency and domain splitting to this octagonal representation. In this paper, we show how abstract interpretation and constraint programming techniques can complement each other for the static analysis of floating-point programs.

5 rAiCp, a hybrid approach

The approach we propose here is based on successive explorations and merging steps. More precisely, we call Fluctuat to compute a first approximation of the variable values at the first program node of the CFG where two branches join. Then, we build one constraint system per branch and use filtering techniques to reduce the domains of the variables computed by Fluctuat. Reduced domains obtained for each branch are merged and exploration goes on with the result of the merge.

5.1 Control flow graph exploration

The CFG of a program is explored using a forward analysis going from the beginning to the end of the program. Statements are converted into DSA (Dynamic

Single Assignment) form where each variable is assigned exactly once on each program path [2]. Lengths of the paths are bounded since loops are unfolded a bounded number of times, after which they are abstracted by the domains computed by abstract interpretation. At any point of an execution path, the possible states of a program are represented by a constraint system over the program variables. Domains of the variables are intervals over the real numbers in the constraint store of RealPaver; domains are intervals over the floating-point numbers that correspond to the `int`, `float` and `double` machine types of the C language[6] in the constraint store of FPCS. Each program statement adds new constraints and variables to these constraint stores. This technique for representing programs by constraint systems was introduced for bounded verification of programs in CPBPV [7]. The implementation of the approach proposed in this paper relies on libraries developed for CPBPV.

CFG exploration is performed on-the-fly and unreachable branches are interrupted as soon as an inconsistency is detected in the constraint store. We collect constraints between two join points in the CFG. If, for all executable paths between these points, the constraint systems are inconsistent for some interval I of an output variable x, then we can remove the interval I from the domain of x. Note that we differentiate between program *input* variables, whose domains cannot be reduced, and program *output* variables, whose domains depend on the program computations and input variable domains, and thus can be reduced.

Merging program states at each join point not only allows a tight cooperation between Fluctuat and the constraint solvers but also limits the number of executable paths to explore.

5.2 Filtering techniques

We use constraint filtering techniques for two different purposes in rAiCp:

- elimination of unreachable branches during CFG exploration;
- reduction of the domain of the variables at CFG join points.

On floating-point numbers constraint systems, we perform $3B(w)$-consistency filtering with FPCS; on real numbers constraint systems, we perform a $BC5$-consistency filtering in paving mode with RealPaver[7].

6 Experiments

In this section, we compare in detail Fluctuat and rAiCp on programs that are representative of Fluctuat limitations. We also compare rAiCp to a state-

[6] Note that the behavior of programs containing floating-point computations may vary with the programming language or the compiler, but also, with the operating system or the hardware architecture. We consider here C programs, compiled with GCC without any optimization option and intended to be run on an x86 architecture managed by a 32-bit Linux operating system.

[7] $BC5$-consistency is a combination of interval Newton method, hull-consistency and box-consistency.

Table 2. Domains of the roots of the `quadratic` function.

		conf. #1: $a \in [-1, 1]$ $b \in [0.5, 1]$ $c \in [0, 2]$			conf. #2: $a, b, c \in [1, 1 \times 10^6]$		
		`x0`	`x1`	Time	`x0`	`x1`	Time
$\mathbb{R}$	Fluctuat	$[-\infty, \infty]$	$[-\infty, \infty]$	0.14 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	0.14 s
	rAiCp	$[-\infty, 0]$	$[-8.006, \infty]$	1.55 s	$[-1 \times 10^6, 0]$	$[-5.186 \times 10^5, 0]$	0.58 s
$\mathbb{F}$	Fluctuat	$[-\infty, \infty]$	$[-\infty, \infty]$	0.13 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	0.13 s
	rAiCp	$[-\infty, 0]$	$[-8.125, \infty]$	0.39 s	$[-1 \times 10^6, 0]$	$[-3\,906.26, 0]$	0.39 s

of-the-art tool, CDFL on the benchmarks provided by the authors of the latter system.

All results were obtained on an Intel Core 2 Duo at 2.8 GHz with 4 GB of memory running Linux using Fluctuat version 3.8.73, RealPaver version 0.4 and the downloadable version of CDFL. All the programs are available at `http://users.polytech.unice.fr/~rueher/Benchs/RAICP`.

6.1 Improvements over Fluctuat

We show here how our approach improves the approximations computed by Fluctuat on programs with conditionals, non-linearities, and loops.

Conditionals: The first benchmark concerns conditional statements, for which abstract domains need to be intersected with the condition of the conditional statement. The function `gsl_poly_solve_quadratic` comes from the GNU scientific library and contains many of these conditional statements. It computes the real roots of a quadratic equation ax^2+bx+c and puts the results in variables `x0` and `x1`.

Table 2 shows analysis times and approximations of the domains of variables `x0` and `x1` for two configurations of the input variables. The first two rows present the results of Fluctuat and rAiCp (with RealPaver) over the real numbers. The next two rows present the results of Fluctuat and rAiCp (with FPCS) over the floating-point numbers.

In the first configuration, Fluctuat's over-approximation is so large that it does not give any information on the domain of the roots, whereas rAiCp drastically reduce these domains both over $\mathbb{R}$ and $\mathbb{F}$. However, intersection of abstract domains has not always such a significant impact on the bounds of all domains as illustrated by the domain over $\mathbb{F}$ of `x0` in the second configuration.

To increase analysis precision, Fluctuat allows to divide the domains of at most two input variables into a given number of sub-domains. Analyses are then run over each combination of sub-domains and the results are merged. Finding appropriate subdivisions of the domains is a critical issue: subdividing may not improve the analysis precision, but it always increases the analysis time. Table 3 reports the results with 50 subdivisions when only one domain is divided, and 30 when two domains are divided. Over $\mathbb{R}$, in the first configuration, the subdivisions

Table 3. Domains over $\mathbb{F}$ for the `quadratic` function with input domains subdivided.

	conf. #1		conf. #2	
	`x0`	Time	`x1`	Time
FLUCTUAT a subdivided	$[-\infty, -0]$	> 1 s	$[-1 \times 10^6, 0]$	> 1 s
FLUCTUAT b subdivided	$[-\infty, \infty]$	> 1 s	$[-5 \times 10^5, 0]$	> 1 s
FLUCTUAT c subdivided	$[-\infty, \infty]$	> 1 s	$[-1 \times 10^6, 0]$	> 1 s
FLUCTUAT a & b subdivided	$[-\infty, -0]$	> 10 s	$[-1.834 \times 10^5, 0]$	> 10 s
FLUCTUAT a & c subdivided	$[-\infty, -0]$	> 10 s	$[-1 \times 10^6, 0]$	> 10 s
FLUCTUAT b & c subdivided	$[-\infty, \infty]$	> 10 s	$[-5 \times 10^5, 0]$	> 10 s

Table 4. Domains of the return value of `sinus` and `rump` functions.

		`sinus` $x \in [-1, 1]$		`rump` $x \in [7 \times 10^4, 8 \times 10^4]$ $y \in [3 \times 10^4, 4 \times 10^4]$	
		Domain	Time	Domain	Time
$\mathbb{R}$	FLUCTUAT	$[-1.009, 1.009]$	0.12 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.13 s
	RAICP	$[-0.842, 0.843]$	0.34 s	$[-1.144 \times 10^{36}, 1.606 \times 10^{37}]$	1.26 s
$\mathbb{F}$	FLUCTUAT	$[-1.009, 1.009]$	0.12 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.13 s
	RAICP	$[-0.853, 0.852]$	0.22 s	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$	0.22 s

yield no improvement and, in the second configuration, the results are identical to those over $\mathbb{F}$.

Subdividing domains can be quite time consuming with little gains in precision:

- In the first configuration, subdivisions of the domain of a lead to a significant reduction of the domain of `x0`. No subdivision combination could reduce the domain of `x1`.
- In the second configuration, the best reduction of the domain of `x1` is obtained by subdividing the domains of both a and b. The gain remains however quite small and no subdivision combination could reduce the domain of `x0`.

RAICP turns out to be more efficient: it often improves the precision of the approximation and requires less time than the subdividing process of FLUCTUAT. Moreover, RAICP could also take advantage of the subdivision technique.

Non-linearity: The abstract domain used by FLUCTUAT is based on affine forms that do not allow an exact representation of non-linear operations: the image of a zonotope by a non-linear function is not a zonotope in general. Non-

Table 5. Domain of the return value of the `sqrt` and `bigLoop` functions.

		sqrt #1: $x \in [4.5, 5.5]$		sqrt #2: $x \in [5, 10]$		bigLoop	
		Domain	Time	Domain	Time	Domain	Time
$\mathbb{R}$	FLUCTUAT	$[2.116, 2.354]$	0.13 s	$[2.098, 3.435]$	0.2 s	$[-\infty, \infty]$	0.15 s
	RAICP	$[2.121, 2.346]$	0.35 s	$[2.232, 3.165]$	0.57 s	$[0, 10]$	0.8 s
$\mathbb{F}$	FLUCTUAT	$[2.116, 2.354]$	0.13 s	$[-\infty, \infty]$	0.2 s	$[-\infty, \infty]$	0.15 s
	RAICP	$[2.121, 2.347]$	0.81 s	$[2.232, 3.168]$	1.59 s	$[0, 10]$	0.7 s

linear operations are thus over-approximated. FPCS handles the non-linear expressions better. This is illustrated on the 7^{th}-order Taylor series of function sinus (see Table 4, column `sinus`).

FPCS and REALPAVER also use approximations to handle non-linear terms, and thus, are not always more precise than FLUCTUAT.The second row of Table 4 shows that RAICP could not reduce the domain computed by FLUCTUAT for the `rump` polynomial program [24], a very particular polynomial designed to outline a catastrophic cancellation phenomenon.

Loops: FLUCTUAT unfolds loops a bounded number of times[8] before applying the widening operator of abstract interpretation. The widening operator allows to find a fixed point for a loop without unfolding it completely. In RAICP, we also unfold loops a user-defined number of times, after which the loop is abstracted by the invariant computed by abstract interpretation. Note that we can also use FLUCTUAT to estimate an upper bound on the number of necessary unfoldings [16].

`sqrt` is a program based on the so-called Babylonian method that computes an approximate value, with an error of 1×10^{-2}, of the square root of a number greater than 4. For the analysis of this program with two different input domains (see Table 5), ten unfoldings are sufficient to exit the loop. Both FLUCTUAT and RAICP obtain accurate results over $\mathbb{R}$. Over $\mathbb{F}$, in the second configuration RAICP shrinks the domain to $[2.232, 3.168]$ whereas FLUCTUAT couldn't achieve any reduction.

Program `bigLoop` contains non-linear expressions followed by a loop that iterates one million times. On such programs, it is not possible to completely unfold loops. FLUCTUAT fails to analyze accurately the loop in this program because of over-approximations of the non-linear expressions. RAICP refines significantly the over-approximations computed by FLUCTUAT, even without any initial unfoldings. This example shows that a tight cooperation between CP and AI techniques can be very efficient.

Contributions of AI and CP FLUCTUAT often yields a first approximation that is tight enough to allow efficient filtering with partial consistencies. Even though the same domain reductions can sometimes be achieved without starting

[8] Default value is ten times.

from the approximation computed by FLUCTUAT (i.e., starting from $[-\infty, \infty]$), our experiments show that our approach usually benefits from the approximation computed by FLUCTUAT.

$3B$-consistency filtering works well with FPCS. $2B$-consistency is not strong enough to reduce the domains computed by FLUCTUAT whereas a stronger kB-consistency is too time-consuming. We experimented also with various consistencies implemented in REALPAVER: $BC5$, a combination of hull and box consistencies with interval Newton method, $HC4$, $3B$-consistency. $3B$-consistency was in general too time-consuming. $BC5$-consistency provided the best trade-off between time cost and domain reduction.

6.2 Comparison with CDFL

CDFL [12] is a program analysis tool designed for proving the absence of runtime errors in critical programs. In [12], the authors show that CDFL is much more efficient than CBMC and much more precise than ASTRÉE [8] for determining the range of floating-point variables on various programs.

We compare here RAICP and CDFL on the set of benchmarks[9] proposed in [12]. The set consists of 57 benchmarks made from 12 programs by varying the input variable domains, the loop bounds, and the constants in the properties to check. We discarded two benchmarks as they are related to integer computations which are not the focus of this work. All the programs are based on academic numerical algorithms, except `Sac` which is generated from a Simulink controller model. The program properties are simple assertions on program variable domains.

Table 6 provides the runing time of RAICP, FLUCTUAT and CDFL. RAICP was only run with FPCS since the properties and the programs are both defined over the floating-point numbers.

All three analyses may report false alarms: i.e., they may answer a property is false while it is not. Actually, RAICP and CDFL correctly reported all the 33 true properties. FLUCTUAT gave 11 false alarms that are noted with * in FLUCTUAT columns of Table 6. The domain refinements performed by RAICP successfully eliminated the false alarms produced by FLUCTUAT.

On average, RAICP is 5 times faster than CDFL for the same precision. On some benchmarks, we observe a speed-up factor of 25. On average, RAICP is 2.2 times slower than FLUCTUAT used alone but this is largely compensated by the gain in precision.

7 Conclusion

In this paper, we introduced a new approach for computing tight intervals of floating-point variables of C programs. RAICP, the prototype we developed, relies on the static analyser FLUCTUAT and on FPCS and REALPAVER, two con-

[9] These benchmarks are available at `http://www.cprover.org/cdfpl`

Table 6. Execution times (s) of CDFL, FLUCTUAT and RAICP.

	CDFL	FLUCTUAT	RAICP		CDFL	FLUCTUAT	RAICP
newton.1.1	0.5	0.12	0.62	eps_line1	0.12	0.11	0.28
newton.1.2	1.64	0.13	0.68	muller	0.13	0.11	0.2
newton.1.3	4.6	0.21	1.89	sac.10	2.49	1.25	1.6
newton.2.1	0.95	0.11	1.47	sac.20	2.46	1.38	1.75
newton.2.2	3.44	0.14	0.82	sac.30	2.49	1.39	1.68
newton.2.3	9.32	0.21	1.79	sac.40	2.47	1.38	1.68
newton.3.1	1.95	0.12*	1.3	sac.50	2.46	1.38	1.71
newton.3.2	5.61	0.13	1.13	sac.60	2.48	1.4	1.76
newton.3.3	15.9	0.22	2.35	sac.70	2.46	1.37	1.7
newton.4.1	1.07	0.12	1.74	sac.80	2.48	1.37	1.7
newton.4.2	8.4	0.13	1.82	sac.90	2.47	1.37	1.67
newton.4.3	23.63	0.22	2.49	sine.1	0.68	0.12	0.31
newton.5.1	1.76	0.12	1.83	sine.2	0.96	0.11	0.28
newton.5.2	14.61	0.13*	2.68	sine.3	0.5	0.11	0.28
newton.5.3	38.19	0.23*	4.01	sine.4	7.89	0.12*	0.3
newton.6.1	1.28	0.12	2.15	sine.5	0.68	0.12*	0.23
newton.6.2	2.33	0.13	8.85	sine.6	0.3	0.12*	0.26
newton.6.3	3.59	0.15	4.76	sine.7	0.13	0.12*	0.22
newton.7.1	1.8	0.12	2.23	sine.8	0.08	0.12	0.23
newton.7.2	1.57	0.14	1.59	square.1	0.16	0.12	0.26
newton.7.3	19.45	0.15	1.68	square.2	0.32	0.12	0.25
newton.8.1	0.41	0.11	0.86	square.3	0.7	0.11	0.25
newton.8.2	1.67	0.12	0.88	square.4	1.05	0.12*	0.22
newton.8.3	7.49	0.12	1.05	square.5	0.68	0.12*	0.22
GC4	0.04	0.14	0.23	square.6	0.55	0.11*	0.23
Poly	0.16	0.11	0.23	square.7	0.36	0.12*	0.23
Rump	0.02	0.11	0.21	square.8	0.06	0.12	0.21
Sterbenz	0	0.12	0.2	**Total**	**208.99**	**18.37**	**40.55**

straint solvers which are respectively correct over floating-point and real numbers. So, RAICP can exploit the refutation capabilities of partial consistencies to refine the domains computed by FLUCTUAT.

We showed that RAICP is fast and efficient on programs that are representative of the difficulties of FLUCTUAT (conditional constructs and non-linearities). Experiments on a significant set of benchmarks showed also that RAICP is as precise and faster than CDFL, a state-of-the-art tool for bound analysis and assertion checking on programs with floating-point computations.

This integration of AI and CP works well because often the first approximation of variable bounds computed by AI is small enough to allow efficient filtering with partial consistencies. In the case of FLUCTUAT, sets of affine forms abstract non-linear expressions and constraints. These sets constitute better approximations of linear constraint systems than the boxes used in interval-based constraint solvers. Nonetheless, they are less adapted for non-linear constraint

systems where filtering techniques used in numeric CSP solving offer a more flexible and extensible framework.

Further work concerns a tighter integration of abstract interpretation and constraint solvers, for instance, at the abstract domain level instead of the interval domain level.

Acknowledgments. The authors gratefully acknowledge Sylvie Putot, Éric Goubault and Franck Védrine for their advice and help on using FLUCTUAT.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: IJCAR. LNCS, vol. 6173, pp. 127–141. Springer (2010)
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. Information Processing Letters 93(6), 281–288 (2005)
3. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: 18th IEEE Symposium on Computer Arithmetic. pp. 187–194. IEEE (2007)
4. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16(2), 97–121 (2006)
5. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: 9th International Conference on Formal Methods in Computer-Aided Design. pp. 69–76. IEEE (2009)
6. Codognet, P., Filé, G.: Computations, abstractions and constraints in logic programs. In: International Conference on Computer Languages (ICCL'92). pp. 155–164. IEEE (1992)
7. Collavizza, H., Rueher, M., Hentenryck, P.V.: A constraint-programming framework for bounded program verification. Constraints Journal 15(2), 238–264 (2010)
8. Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzers: A comparison with ASTRÉE. In: 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. pp. 3–20. IEEE (2007)
9. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: FMICS. LNCS, vol. 5825, pp. 53–69. Springer (2009)
10. Denmat, T., Gotlieb, A., Ducassé, M.: An abstract interpretation based combinator for modeling while loops in constraint programming. In: Principles and Practices of Constraint Programming (CP'07). LNCS, vol. 4741, pp. 241–255. Springer Verlag (2007)
11. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Transactions on Computers 60(2), 242–253 (2011)
12. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric bounds analysis with conflict-driven learning. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7214, pp. 48–63. Springer (2012)
13. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: CAV. LNCS, vol. 6174, pp. 212–226. Springer (2010)
14. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)

15. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: SAS. LNCS, vol. 4134, pp. 18–34. Springer (2006)
16. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. LNCS, vol. 6538, pp. 232–247. Springer (2011)
17. Granvilliers, L., Benhamou, F.: Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software 32(1), 138–156 (2006)
18. Harrison, J.: A machine-checked theory of floating-point arithmetic. In: TPHOLs. LNCS, vol. 1690, pp. 113–130. Springer-Verlag (1999)
19. Lhomme, O.: Consistency techniques for numeric CSPs. In: 13th International Joint Conference on Artificial Intelligence. pp. 232–238 (1993)
20. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: CP. LNCS, vol. 6308, pp. 360–367. Springer (2010)
21. Michel, C.: Exact projection functions for floating-point number constraints. In: 7th International Symposium on Artificial Intelligence and Mathematics (2002)
22. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: CP. LNCS, vol. 2239, pp. 524–538. Springer Verlag (2001)
23. Pelleau, M., Truchet, C., Benhamou, F.: Octagonal domains for continuous constraints. In: Principles and Practice of Constraint Programming (CP'11). LNCS, vol. 6876, pp. 706–720 (2011)
24. Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica 19, 287–449 (2010)